Checking the TWIN Elevator System by translating Object-Z to SMV

Sören Preibusch[1] and Florian Kammüller[2]

[1] German Institute for Economic Research
Mohrenstraße 58, 10117 Berlin
spreibusch@diw.de
[2] Technische Universität Berlin
Fakultät IV: Elektrotechnik und Informatik
Franklinstraße 28-29, 10587 Berlin
flokam@cs.tu-berlin.de

Abstract. In the context of large scale industrial installations, model checking often fails to tap its full potential because of a missing link between a system's specification and its functional and non-functional requirements, like safety. Our work bridges this gap by providing a translation from the formal specification language Object-Z to the SMV model checker input language to combine their advantages.
This paper focuses on the translation of the object-oriented features of Object-Z: operation promotion and communication between objects. We demonstrate the feasibility of our approach using the example of the TWIN Elevator system and embed the translation process in the industrial software production workflow.

1 Introduction and Related Work

Software development for industrial purposes differs from application development by the nature of the constructed software products and by the nature of the production process. Industrial software enables the effective and efficient usage of large installations and equipment in aviation, power generation, logistics, medical treatment, and production lines. These systems are typically safety-critical; disturbance of their well-functioning may cause personal or physical damage.

Model checking techniques are used to check properties of these systems; they provide reliable results by including a system's whole state space in mathematical proofs of correctness.

A variety of model checking tools has emerged along with different input languages. As a standardized input format does not exist yet, interoperability between users and re-use of specification is hampered. The lack of established authoring tools and intuitive means to structure large specifications are additional drawbacks. The ability to use Object-Z as a common input language would allow to overcome these difficulties. Object-Z [4] is an object-oriented extension of the standardized specification language Z [5]. It has well understood semantics [13] and benefits from tool support [2], Section 5.

Advantages of Combining Object-Z and SMV. Coupling Object-Z as a system specification language with model checking support manifests advantages when compared to purely verifying Z specifications [19]. These advantages originate the specification phase and the checking phase in the workflow.

Whereas a Z specification defines a single state space, Object-Z's classes with their separate namespaces are especially handy for specifying medium- to large-scale software systems [12]. The object-oriented specification paradigm is well adapted to distributed and embedded systems; communicating objects reflect the spatial separation of different components. Unlike Z, Object-Z supports specifying concurrent systems. Multiple instantiation of the same class provides for easy scalability where Z would have required a manual enumeration of each instance.

Moreover, translating Object-Z to SMV enables the use of general-purpose model checking tools. Those profit from a larger community and ongoing research resulting in performance enhancements.

Finally, there is a difference between Z and SMV in the nature of the properties that may be expressed (and thus checked). Z and Object-Z specifications are limited to first order predicate logic whereas SMV is designed for temporal logic expressed in CTL or LTL formulas. Those temporal formulas are naturally checked against the specification; in contrast, Z checkers usually only perform type checks or well-formedness checks.

Previous Work has provided model checking support for the base language Z [14]. However, its authors have seen the extension to Object-Z as future work. Especially the object-oriented features make this a non-trivial task. [18] describes a translation procedure from Object-Z to SMV using ASM as an intermediate language. Then again, this works lacks considering the semantics of an Object-Z specification as a description of a system embedded in an environment. Hence, the translation of operations is problematic. Inter-object communication is hard to follow and distributed operations operators are not covered. Moreover, that work does not preserve the structure of an Object-Z specification but instead flattens the top-level structure provided by classes and modules.

Our contribution is twofold. First, on a concrete level, we present a specification of the TWIN elevator system in the formal specification language Object-Z. We provide a step-by-step translation to an equivalent (Cadence) SMV program [9]. Second, on an abstract level, we elaborate general rules for the translation process, focusing on the object-oriented features of Object-Z: operation promotion and communication between objects. In addition, we sketch how the translation can be integrated rewardingly in the workflow of industrial software production processes.

The remainder of this paper is organized as follows. The following Section portrays the TWIN elevator system by ThyssenKrupp. Section 3 is the core of the paper. It presents the commented TWIN's specification in Object-Z along with the SMV equivalents and general translation rules. The resulting SMV program is enriched by temporal formula stating fairness and safety requirements that are

successfully checked. Section 5 embeds the translation process in the industrial software development process prior to concluding in Section 6 with a summary and outlook.

2 TWIN Elevator System Case Study

The idea of having an elevator with two independent cabins operating in the same shaft dates back to the 1930s. However, first attempts to build this efficient transportation system failed and the engineering of a control system has been an unsolved problem for almost a century. Only in 2002 ThyssenKrupp installed the first TWIN elevator system at Stuttgart University.

In a TWIN elevator system, two cabins are arranged one above the other; they run independently in the same TWIN shaft – also at different speeds. A safety distance is kept, depending on the speeds involved. The cabins can move in different directions, which means that they can also move toward each other [17]. Because the TWIN cabins cannot sidestep, each TWIN installation comprises at least one conventional shaft to serve routes that would result in a crossing of the TWIN cabins (Fig. 1).

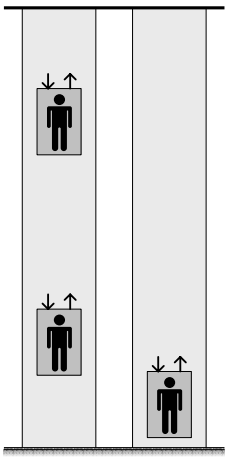


Fig. 1. Minimal TWIN installation (schematic view): a TWIN shaft with two cabins on the left and conventional shaft on the right

A prospective passenger communicates his destination level no longer within the elevator cabin, but instead by Destination Selection Control (DSC) terminals mounted on each floor. The control system then selects one of the cabins capable to serve the call.

The informal specification of safety requirements of ThyssenKrupp has been the basis for their formal expression by means of formal specification and model checking [7]. In [7], we developed a detailed SMV program to check the TWIN's well-functioning and provide evidence for the scalability of model checking procedures. However, the crafting of an SMV program that large is unrealistic to be carried out in an industrial context. In contrast, it is more likely that Object-Z specifications are used and developed already in an early project stage.

The earlier results also act as a benchmark for our translation process in that applying model checking on an SMV program resulting from an automated translation should not perform worse than on the hand-made SMV program.

In addition, the duo of this paper and the first TWIN case study is an example for abstraction. The TWIN specification developed in the next section is just detailed enough to examine fairness and safety requirements.

3 Translating Object-Z Specifications to SMV Programs

3.1 Fundamental Object-Z concepts

In Object-Z, graphical *schema* notation enables the concise structuring of state and operation specifications and modularizes them into classes. Any schema consists of a declaration part and a predicate part enabling abstract specification of invariants, pre-conditions and post-conditions. Classes in Object-Z encapsulate a state and an initial schema, as well as operation schemas specifying the methods of an object oriented class. In addition, Object-Z features specific class constructs for visibility, constant declarations, polymorphism, and inheritance.

The idea of instantiation of an object o of a class C is naturally represented by the declaration of a variable $o : C$ where o then denotes the identity of an object. Object-Z has a reference semantics [13] and the common object-oriented dot notation, e.g. $o.m$ to annotate the invocation of an object's feature.

The so-called *schema calculus* comprises operators enabling composition of operations to create new operations, especially in the context of modular systems. In Object-Z operations are composed by conjunction $\wedge$, non-deterministic choice $[]$, sequential composition $\fatsemi$, and parallel composition $\|$.

In Section 3.3, we will piecewise present the TWIN's specification in *Object-Z* along with explanations of the newly introduced Object-Z features. We outline the corresponding translation rule and present the (one or more) resulting `SMV code` fragments. We have partitioned large classes; the splits are clearly signed (. . . / [cont'd]). Where appropriate, we skip over specification parts that would not contribute to introduce new translation rules. In addition to this paper, the unsplit and unabridged versions are available online [10],[11].

3.2 Directness and Structure Preservation

Our translation from Object Z to SMV is *direct* in that it identifies concepts of Object-Z, like propositional logic, basic types, and the class concept, with almost directly corresponding features of SMV. Where appropriate, the missing semantics is added in the translation process using additional definitions, constraints, or other constructions as we will see. The striking advantage of this direct translation is that it is quite obviously *structure preserving*, i.e. the structure of the Object-Z classes and SMV modules correspond one to one and the initial and state schemas of Object-Z have distinct representations in SMV code chunks. Although the granularity of the operations cannot be preserved, one can show that the translation distributes over the constructs of SMV used for operation representation.

3.3 Translation Rules

Type Definitions and Constants. Types and constants used within the Object-Z specification are defined at its beginning. Types can be defined by

enumerating their values or as an integer sub-range. We define a type for the cabin status and for the storeys.

Expressions in the constant definition are evaluated once during the translation process; static evaluation is correct as – by definition – Object-Z constants do not change their values. Definitions for the boolean constants are added. In SMV, truth values are represented by integers.

$CabinStatus ::= \text{vacant} \mid \text{busy}$
$Level ::= (1 \,..\, 12)$

$LevelGround = min\ Level$
$LevelTop = max\ Level$

```
typedef CabinStatus {vacant, busy};
typedef Level 1..12;

#define LevelGround 1
#define LevelTop 12

#define true 1
#define false 0
```

Classes. Following the object-oriented paradigm, classes are the top-level structuring mechanism in Object-Z. They provide a scope for variables and may contain operations that change the variables' values by state transition. Our specification comprises four classes: A *Call* class, acting as a datatype for calls with the attributes *from* and *to* coding the route's endpoints, a *Cabin* class for cabins in a conventional shaft or in a TWIN elevator shaft, a class for the *DSC*, and a class for the *TWIN_System* itself.

A class' state variables are noted inside an Object-Z box. Variables are typed and can instantiate classes. In SMV, modules provide a similar scoping mechanism.

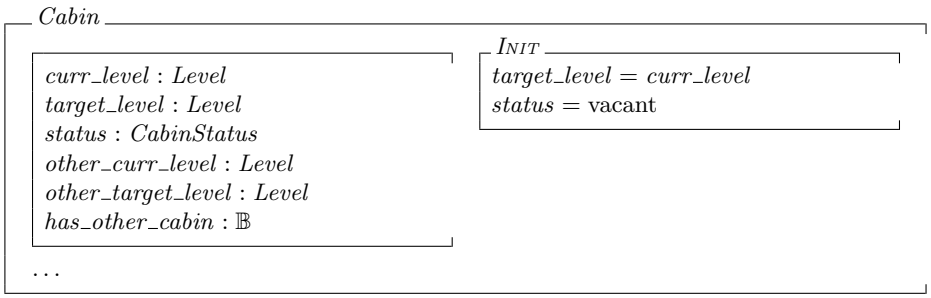
Call

$from : Level$
$to : Level$

```
module Call() {
  from : Level;
  to : Level; }
```

Initial schema. An Object-Z class can include an *INIT* schema, assembling predicates that must hold in the initial state. Initially, a cabin is vacant and its target level is the current level so that there is no induced call. The current level is initialized upon instantiation in the *TWIN_System* class. Cabins in a TWIN shaft have *has_other_cabin* set to true and the variables *other_curr_level* and *other_target_level* referring to the other cabin in the same shaft. Therefore, a TWIN cabin is aware of the other cabin's position – an information needed when deciding whether the cabin may accept a call or not.

The predicates over the initial state are translated to an active initialisation in SMV.




```
module Cabin() {
  curr_level : Level;
  target_level : Level;
  status : CabinStatus;
  other_curr_level : Level;
  other_target_level : Level;
  has_other_cabin : boolean;

  init(target_level) := curr_level;
  init(status) := vacant;
```

State Transitions: Precondition and Stimulus. In Object-Z, state transitions are realized by named operations that change the values of the state variables enumerated in their Δ-lists. Below a horizontal line, predicates over the variables' values before the state transition are noted (precondition of the operation). The primed variable names refer to the variables' values after the operation's execution (postcondition).

The operations `MoveUp` and `MoveDown` realize the state transition of the cabin with regard to its current level. The operations' preconditions assure that the cabin moves in the direction of its target level and does not run out of the shaft.

For each operation, we introduce two defined boolean variables in the SMV program. These variables do not add to the state vector and thus do not impact on the performance of verification. The SMV variable *`operationname`*`_pre` has the truth value of the precondition. It is hereby also a translation of the Object-Z expression "pre *operationname*" that represents the truth value of the operation's precondition.

The variable *`operationname`*`_stimulus` indicates whether there is a call of the operation from the environment. According to the semantics of Object-Z [12], the specified system is embedded in an environment that may evoke an operation. Unless this evokation occurs, the state transition specified by the operation does not take place. This is in contrast to SMV, where each possible state transition is executed. Hence, the variable *`operationname`*`_stimulus` acts as an additional guard.

Cabin [cont'd]

...

MoveUp

$\Delta(curr\_level)$

$curr\_level < LevelTop$
$curr\_level < target\_level$
$curr\_level' = curr\_level + 1$

MoveDown

$\Delta(curr\_level)$

$curr\_level > LevelGround$
$curr\_level > target\_level$
$curr\_level' = curr\_level - 1$

...

```
/* operation MoveUp */
MoveUp_pre : boolean;
MoveUp_pre := (curr_level < LevelTop) & (curr_level < target_level);

MoveUp_stimulus : boolean;


/* operation MoveDown */
MoveDown_pre : boolean;
MoveDown_pre := (curr_level > LevelGround) & (curr_level > target_level);

MoveDown_stimulus : boolean;
```

All operations possibly changing a state variable can be identified by examining their Δ-lists. For each state variable, the influencing operations are collected; their respective precondition and stimulus variables guard the state transition in SMV.

The general schema has the form:

```
next(variable) := case{
  op1_pre & op1_stimulus : op1_postpredicate;
  op2_pre & op2_stimulus : op2_postpredicate;
  ...
  default : variable; }
```

The last alternative (`default`) results in the variable to remain unchanged if none of the operations is executed.

```
next(curr_level) := case{
  MoveUp_pre & MoveUp_stimulus : curr_level + 1;
  MoveDown_pre & MoveDown_stimulus : curr_level - 1;
  default : curr_level; };
```

Communication variables. Communication variables can be defined in the local scope of an operation. Output variables are decorated with an exclamation mark, input variables with a question mark. Communication variables in opposite directions with the same basename are identified when operations are combined (see below on page IX).

The cabin's operation *AcceptCall* may record a *new_call?* for the cabin if the cabin is currently vacant (first precondition). In addition, if the cabin is a TWIN cabin (*has_other_cabin* is true), it can only accept the call in case call processing would not result in a crash with the other cabin (second precondition).

If the cabin has accepted the call, its *status* is set to busy and the call's attribute *to* is taken as the cabin's new *target_level*. If the cabin has finished a call, its *status* is set to vacant.

In the translation to SMV, the communication variable *new_call?* is prefixed with `in` (`out` for output communication variables) and with the operation name, to provide for a local scope.

Cabin [cont'd]

...

AcceptCall

$\Delta(status, target\_level)$
$new\_call? : Call$

$has\_other\_cabin = \text{false} \lor$
$(new\_call?.to - other\_target\_level) \times$
$(curr\_level - other\_curr\_level) > 0$
$status = \text{vacant}$
$status' = \text{busy}$
$target\_level' = new\_call?.to$

FinishCall

$\Delta(status)$

$curr\_level = target\_level$
$status = \text{busy}$
$status' = \text{vacant}$

...

```
/* operation AcceptCall */
AcceptCall_pre : boolean;
AcceptCall_pre := (has_other_cabin = false) |
  ( (AcceptCall_in_new_call.to - other_target_level) *
    (curr_level - other_curr_level) > 0 ) & (status = vacant) ;

AcceptCall_stimulus : boolean;

AcceptCall_in_new_call : Call;

/* operation FinishCall */
FinishCall_pre : boolean;
FinishCall_pre := (curr_level = target_level) & (status = busy);

FinishCall_stimulus : boolean;

next(target_level) := case{
  AcceptCall_pre & AcceptCall_stimulus : AcceptCall_in_new_call.to;
  default : target_level; };

next(status) := case{
  AcceptCall_pre & AcceptCall_stimulus : busy;
  FinishCall_pre & FinishCall_stimulus : vacant;
  default : status; };
```

Operation Promotion and Communication. Operations defined by operation schemas may be used to define new operations by composition. These "operation promotions" are placed inside a class. A new operation *op* can be defined by:

- conjunction: $op \mathrel{\widehat=} op1 \wedge op2$
  both $op1$ and $op2$ are executed
- (non-deterministic) choice: $op \mathrel{\widehat=} op1 \,[\!]\, op2$
  one of $op1$ and $op2$ is arbitrarily chosen and executed. If the precondition of one of the compounding operations is not fulfilled, the operation is removed from the choice.
- parallel composition: $op \mathrel{\widehat=} op1 \parallel op2$
  both $op1$ and $op2$ are executed with bi-directional communication
- sequential composition: $op \mathrel{\widehat=} op1 \fatsemi op2$
  both $op1$ and $op2$ are executed with forward communication only

The operators for operation composition can be combined and several operations can be combined at once.

Communication between operations is realized by matching the communication variables. Bi-directional communication means that the values of communication variables with the same basename are identified. Forward communication means that only the output variables of the first operation are matched with the input variables of the second operation. An operation lacking communication variables does not participate in communication. In general, communication variables need not match; the unmatched communication variables of the composed operations are then simply unified in the signature of the combined operation. In the case of the choice operator, the unified signatures of the involved constituent operations must be identical.

Operation Promotion: Choice. We define a new operation *Move* as the choice between the operations *MoveUp*, *MoveDown*, and *FinishCall* depending on whether the cabin's current level is below, above, or equal its target level. In case the current level equals the target level, the call has been processed.

Analogously to operations defined by operation schemas, two boolean variables for stimulus and precondition are introduced in the translation. The precondition of the promoted operation is calculated by combining the preconditions of the compounding operations (see Table 1).

Operator	Precondition escalation (SMV)	Stimulus propagation	Communication
$\wedge$	conjunction (&)	conjunction	none
$[\!]$	disjunction (|)	exclusive disjunction	none
$\parallel$	conjunction (&)	conjunction	bi-directional
$\fatsemi$	conjunction (&)	conjunction	forward

Table 1. Operation operators overview

Cabin [cont'd]

$\ldots$

$Move \mathrel{\widehat=} MoveUp \,[]\, MoveDown \,[]\, FinishCall$

```
Move_pre : boolean;
Move_pre := MoveUp_pre | MoveDown_pre | FinishCall_pre;

Move_stimulus : boolean;
```

The non-deterministic choice between two operations susceptible to be chosen (i.e. whose preconditions evaluate to true) is realized in SMV by assigning a set of values to a variable. This assignment is understood as that one value of the set is arbitrarily chosen each time and assigned to the variable.

We use SMV's construct of guarded set membership when enumerating the set elements: `cond ? elem` means that `elem` is included in the set if `cond` evaluates to true.

```
Move_choice : {1,2,3};
Move_choice := {
  (MoveUp_pre) ? 1,
  (MoveDown_pre) ? 2,
  (FinishCall_pre) ? 3 };
```

The stimulus from the promoted operation propagates to the compounding operations as defined in Table 1. The arbitrary choice between the set values assures that the stimulus propagates to only one of the compounding operations:

```
MoveUp_stimulus :=
  (Move_stimulus & Move_choice = 1);

MoveDown_stimulus :=
  (Move_stimulus & Move_choice = 2);

FinishCall_stimulus :=
  (Move_stimulus & Move_choice = 3);
```

Recapitulative Example: the *DSC* class. So far we know how to translate Object-Z classes, state variables, operations, and communication variables to SMV. We now apply these rules to translate the small *DSC* class.

The Destination Selection Control (DSC) terminal registers the passenger's ride request. The calls are communicated to the cabins; the storey where the DSC is mounted (*location*) is the call's *from* attribute. The translation to SMV follows the principles established above.

DSC

$location : Level$

PlaceCall

$\Delta()$

$new\_call! : Call$

$new\_call!.from = location$

```
module DSC() {

  /* state variables */                  /* operation PlaceCall */
  location : Level;                      PlaceCall_pre : boolean;
                                         PlaceCall_pre := true;
  PlaceCall_stimulus : boolean;

  PlaceCall_out_new_call : Call ;
  PlaceCall_out_new_call.from := location; }
```

Multiple Instantiation. The class *TWIN_System* models the TWIN elevator system. It instantiates the previously defined class *Cabin* thrice – once for a conventional cabin and twice for the TWIN cabins. In each storey, a DSC is mounted, resulting in a functional mapping from a *Level* to a *DSC* object.

TWIN_System

$dscs : Level \rightarrow DSC$
$twin\_lower, twin\_upper, conventional : Cabin$

$\forall\, l \in \mathrm{dom}\, dscs \bullet dscs(l).location = l$
...

...

```
module TWIN_System() {

  /* state variables */
  dscs : array Level of DSC ;
  forall(l in Level)
    dscs[l].location := l;

  twin_lower, twin_upper, conventional : Cabin;
```

Operation Promotion: Conjunction. We combine the *Move* operations of all cabins in the *TWIN_System* to a single operation *MoveCabins*. Since all cabins move independently, we use an operation operator without communication but with conjunctive stimulus propagation: the 'and' operator $\wedge$.

TWIN_System [cont'd]

...
$MoveCabins \mathrel{\widehat{=}} twin\_lower.Move \wedge twin\_upper.Move \wedge conventional.Move$
...

```
  /* operation promotion MoveCabins */
  MoveCabins_pre : boolean;
  MoveCabins_pre := twin_lower.Move_pre | twin_upper.Move_pre |
    conventional.Move_pre;
```

```
MoveCabins_stimulus : boolean;

twin_lower.Move_stimulus := MoveCabins_stimulus;
twin_upper.Move_stimulus := MoveCabins_stimulus;
conventional.Move_stimulus := MoveCabins_stimulus;
```

A priori, a call may be processed by any of the available cabins. Any of the cabins capable of processing the call may accept it (see the operation *AcceptCall* in the *Cabin* class). Therefore, we use a non-deterministic choice $[]$. The call accepting by the cabins occurs in parallel ($\|$) with the call placing by the DSCs. We use a 'distributed' choice ($\bigsqcap$) between the *PlaceCall* operations of all *DSC* objects.

Distributed Operation Promotion. The distributed choice operator in Object-Z provides a non-deterministic choice over a range of objects whose methods are enabled. In SMV, the distributed precondition escalation is realized by applying the boolean precondition combination operator over an array constructed of all individual operations' preconditions: `f[expr(var) : var in Type]` applies the operator `f` (`|` or `&` according to Table 1) distributively over all expressions `expr(var)`.

The set of choice alternatives is constructed analogously by using SMV's iterative construction capabilities: the expression `expr(var)` is included in the set {`expr(var) : var in Type, cond(var)`} if `cond(var)` evaluates to true.

Finally, the stimulus propagation iterates over all DSC object's *PlaceCall* operations indexed by `l in Level`. We observe another structure preservation property in that the distributivity is preserved and an explicit enumeration of all *DSC* instances is not necessary during the translation process.

TWIN_System [cont'd]

$$
\begin{array}{l}
\ldots \\
DistributeCalls \mathrel{\widehat=} \bigsqcap l : Level \bullet dscs(l).PlaceCall \parallel \\
\quad (twin\_lower.AcceptCall \;[]\; twin\_upper.AcceptCall \;[]\; conventional.AcceptCall) \\
\ldots
\end{array}
$$

```
/* operation promotion DistributeCalls */
DistributeCalls_pre : boolean;
DistributeCalls_pre := |[dscs[l].PlaceCall_pre : l in Level ] &
  (twin_lower.AcceptCall_pre | twin_upper.AcceptCall_pre |
  conventional.AcceptCall_pre);

DistributeCalls_stimulus : boolean;

DistributeCalls_choice : {1,2,3};
DistributeCalls_choice := {
  (twin_lower.AcceptCall_pre) ? 1,
  (twin_upper.AcceptCall_pre) ? 2,
  (conventional.AcceptCall_pre) ? 3 };
```

```
DistributeCalls_choice_2 : Level;
DistributeCalls_choice_2 := { l : l in Level, dscs[l].PlaceCall_pre};

twin_lower.AcceptCall_stimulus :=
  (DistributeCalls_stimulus & DistributeCalls_choice = 1);

twin_upper.AcceptCall_stimulus :=
  (DistributeCalls_stimulus & DistributeCalls_choice = 2);

conventional.AcceptCall_stimulus :=
  (DistributeCalls_stimulus & DistributeCalls_choice = 3);

forall(l in Level)
  dscs[l].PlaceCall_stimulus :=
    (DistributeCalls_stimulus & DistributeCalls_choice_2 = l);
```

Matching Communication Variables. The parallel operator $\|$ results in a communication between the chosen *PlaceCall* operation and the chosen *AcceptCall* operation as described on page IX: `AcceptCall_in_new_call` is assigned the value of `PlaceCall_out_new_call` in case the *DistributeCall* operation is stimulated. The communication is conditioned over the choice ($[]$) only with regard to the outputting operation (realized by `DistributeCalls_choice_2` in the translation to SMV). It must not be conditioned with regard to the choice of the receiving operation: because choosability depends on an operation's precondition, which in turn may depend on an input variable, a circular definition would occur.

```
/* operation promotion DistributeCalls - communication */
twin_lower.AcceptCall_in_new_call := case {
  DistributeCalls_stimulus :
    dscs[DistributeCalls_choice_2].PlaceCall_out_new_call; };

twin_upper.AcceptCall_in_new_call := case {
  DistributeCalls_stimulus :
    dscs[DistributeCalls_choice_2].PlaceCall_out_new_call; };

conventional.AcceptCall_in_new_call := case {
  DistributeCalls_stimulus :
    dscs[DistributeCalls_choice_2].PlaceCall_out_new_call; };
```

Finally, we combine the TWIN system's two tasks (cabin movement and call management) in a single *Operate* operation.

TWIN_System [cont'd]

...

$Operate \mathrel{\widehat{=}} MoveCabins \wedge DistributeCalls$

```
  /* operation promotion Operate */
  Operate_pre : boolean;
  Operate_pre := true;

  Operate_stimulus : boolean;

  MoveCabins_stimulus := Operate_stimulus;
  DistributeCalls_stimulus := Operate_stimulus;
```

Adding the `main` module. SMV requires one `main` module in each program. All top-level modules are instantiated once in this module. Also, the stimulus for all operations not used to construct any other operation inside the Object-Z specification is set to true to assure that a 'running' system is checked. As we aim at an automated mechanical translation procedure, it is noteworthy that these operations are easy to enumerate.

```
module main() {
  system : TWIN_System();
  system.Operate_stimulus := true; }
```

4 Model Checking the Translation with SMV

After the Object-Z specification has been translated in SMV, one can enrich the program by (temporal) formulas expressing crucial system requirements. As an illustration, we express requirements regarding fairness, correct call processing, and safety in SMV:

For each cabin, the `Fairness` properties state that a call will be finished: always, if the cabin is busy, it will be vacant in the future:

```
  Fairness_1 : assert
    G (system.twin_lower.status = busy) ->
      F (system.twin_lower.status = vacant);
  Fairness_2 : assert
    G (system.twin_upper.status = busy) ->
      F (system.twin_upper.status = vacant);
  Fairness_3 :
  assert
    G (system.conventional.status = busy) ->
      F (system.conventional.status = vacant);
```

The `Processing` properties assure that the call termination is only achieved if the cabin really reaches its target level.

```
  Processing_1 : assert  G (system.twin_lower.status = busy)
      U (system.twin_lower.curr_level = system.twin_lower.target_level);
  Processing_2 : assert  G (system.twin_upper.status = busy)
      U (system.twin_upper.curr_level = system.twin_upper.target_level);
  Processing_3 : assert  G (system.conventional.status = busy)
      U (system.conventional.curr_level = system.conventional.target_level);
```

For the TWIN shaft, the `Safety` property would be violated in case of a crash. It requires the upper TWIN cabin to always stay above the lower TWIN cabin.

```
Safety : assert
  G (system.twin_upper.curr_level > system.twin_lower.curr_level);
```

The assertions are noted in the SMV `main` module, thence prefixed with `system.` to reference the objects. One can place assertions in any module; we opted for the `main` module to emphasize on the separation between the specified system and the requirements towards it.

All properties together were successfully checked within seconds on standard desktop hardware. SMV allocated 34881 BDD nodes. Experiments showed that the time necessary to check the properties is only marginally influenced by the number of storeys.

To leverage this verification potential during the software development process, we have automated the translation process in a web-based prototype (called ZOË) and sketch its workflow embedding in the next section.

5 Workflow Embedding

The industrial software development process exhibits specifities that require fitted management and tools. The engineering of complex software systems on a large scale usually involves many developers, possibly from different backgrounds. Tightly coupled heterogenous components in an installation are developed by cross-functional teams. Each of the team members is an expert in one domain of the software's functionality. Documentation along the process is crucial and the documents regularly are one of the manufacturer's deliverables. For this reason, the documents need to be well presented, with appropriate languages and notations, so that they can be understood accurately and used effectively [8].

However, domain-specific language dialects across departments often hamper a holistic documentation and complicate the cooperation inside the development team. Requirements are typically expressed in a different language than the system description (e.g. CTL formula for safety requirements vs. Object-Z as a common language for the functional specification) and inspectors may also be an external party not involved in the software development process.

A unifying formal methods approach can conciliate between these different formalisms and promises advances in product and process quality: the software will better fulfill its requirements and adherence to delivery dates and budget will be improved. Similar endeavors have been undertaken in the Alloy project [16]. The Alloy Analyzer is a tool that checks properties on a model, visualizes, and simulates it. However, the Alloy Analyzer is a 'model finder' that finds any model satisfying a logical formula, rather than checking a formula on an operationally specified model.

Our portrayed translation from an Object-Z specification to a checkable software model bridges the described gap and brings together domain-experts from functionality specification and requirements checking. Functional and non-functional requirements can be checked as demonstrated exemplarily in the previous section. In case an error is found, a counterexample is generated and the detected discrepancy between the required and the actual behaviour can be traced back to the original specification because of the structure-preservation. If, for instance, a property is breached subsequent to a state transition, the operations causing the state transition are enumerated. The manual workload is reduced and applicability of formal methods is thereby extended.

To support the workflow, we have developed a web-based authoring environment for Object-Z specifications, ZOË, see Fig.2. The translation process described in this paper is implemented as a prototype fully integrated with our front end tool ZOË.

A domain-expert can develop the specification inside his browser, and the web-based infrastructure supports collaborative engineering. In contrast to previous tool support, no special software installations or plugins are needed. The editor and the translator are implemented using HTML, CSS and JavaScript – available with any current browser. The editor alleviates the expert's tasks as it allows an interactive specification development. The formulated Object-Z specification can be exported by means of output/formatter plugins.

The checked model can be refined to an executable program, so that end-to-end quality assurance can be achieved [1]. The refinement may be carried out based on the generated SMV program as the translation produces an easily readable output: variable names are maintained and so is the specification's inherent structure.

The on the fly checking of the fairness and safety properties of our TWIN case study provides evidence for applicability also for larger systems.

6 Conclusion and Discussion

We have established and explained rules for translating an Object-Z specification to a corresponding SMV program. These rules make full use of the close correspondence between many important features of Object-Z and SMV whilst being careful not to identify syntactical similarities whose semantics do not match. Besides propositional logic and basic datatypes, one major correspondence we identified is that of prestate and poststate. We cover the object-oriented concepts of Object-Z and non-deterministic choice, and we successfully cope with object communication and operation promotion.

Object-Z is a powerful specification formalism and it is generally not possible to represent it in its entirety in the input language of SMV. However, using simple restrictions of infinite base types and limiting the predicate language to equations and simple quantified expressions, we arrive at a useful subset of the original specification language that can be verified automatically with standard model checking techniques.

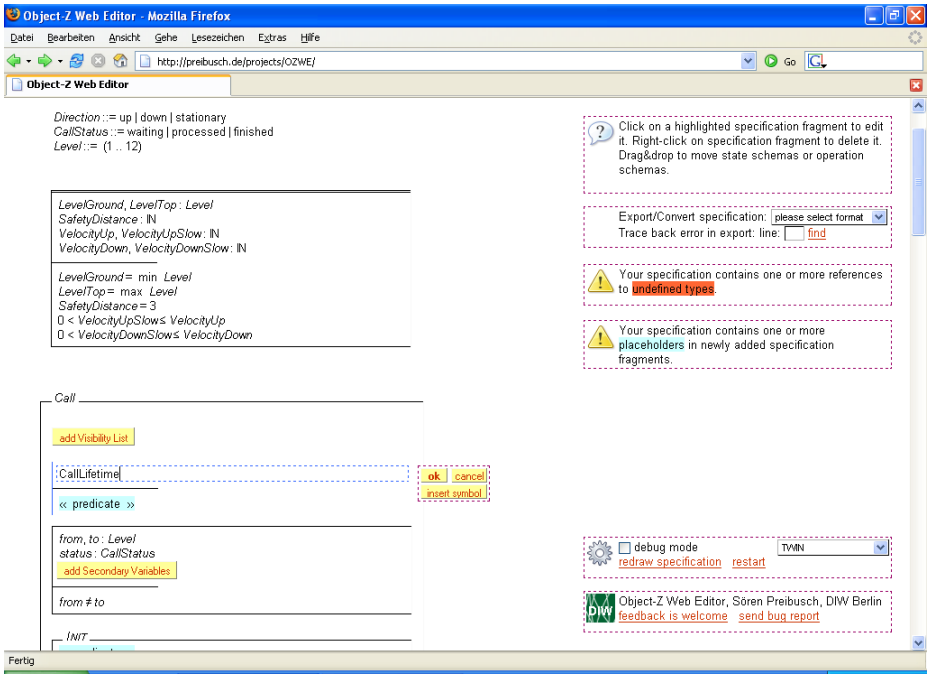


Fig. 2. The ZOË Workspace

The resulting translation enables the further application of model checking techniques to verify the specified system's correct behaviour. We therefore embed the translation procedure in the industrial software production process and have developed appropriate web-based tool support.

Our method's suitability is attested by its application to the cutting-edge TWIN elevator case-study. Starting from a concise system specification in Object-Z, we employ the translation algorithm to optain a checkable TWIN model. Fairness and safety requirements are verified on this model within seconds.

The approach we take in this work is pragmatic but sound. Although soundness is not formally verified, it is fairly evident as we exploit natural similarities between state based specification in Object-Z and the state model of SMV. The integration of temporal logics and Object-Z semantics is furthermore based on well-established results [3], [15].

A very important point in favour of our approach to translating Object-Z into SMV directly, that makes it stand out in comparison to other similar endevours, is its *shallowness* [6]: the concepts of the application are identified in a one-to-one fashion with concepts of the formal target language. Here the application is Object-Z and the formal target language is SMV. The striking advantage of shallowness is that we inherit the full expressiveness of the target language and hence the full power of any available support.

References

1. Tobias Amnell. Code Synthesis for Timed Automata. Thesis, Uppsala University, 2003.
2. The Community Z Tools project, 2006. `http://czt.sourceforge.net/`
3. John Derrick and Graeme Smith. Linear temporal logic and Z refinement. *Algebraic Methodology and Software Technology (AMAST 2004)*. Springer LNCS **3116**, 2004.
4. Roger Duke and Gordon Rose. Formal Object-Oriented Specification Using Object-Z. Cornerstones of Computing. MacMillan, 2000.
5. International Organization for Standardization: ISO/IEC 13568:2002: Information technology – Z formal specification notation – Syntax, type system and semantics. `http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=21573`
6. Florian Kammüller *Interactive Theorem Proving in Software Engineering.* Habilitationsschrift, Technische Universität Berlin, 2006.
7. Florian Kammüller and Sören Preibusch. An Industrial Application of Symbolic Model Checking – The TWIN-Elevator Case Study. Accepted for publication in *Informatik Forschung und Entwicklung.* Springer, 2007.
8. Shaoying Liu. Formal Engineering for Industrial Software Development. Springer, 2004
9. Ken McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1995.
10. Sören Preibusch. TWIN Elevator System, Concise Object-Z Specification, 2007 `http://preibusch.de/projects/TWIN/Concise_OZ`
11. Sören Preibusch. TWIN Elevator System, Concise Object-Z Specification (Translation to SMV), 2007 `http://preibusch.de/projects/TWIN/Concise_OZ_Translation_SMV`
12. Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods, Kluwer Academic Publishers, 2000.
13. Graeme Smith and Florian Kammüller, Thomas Santen. Encoding Object-Z in Isabelle/HOL. *ZB 2002: Formal Specification and Development in Z and B*Springer LNCS **2272**, 2002.
14. Graeme Smith and Luke Wildman. Model Checking Z Specifications Using SAL. *ZB 2005: Formal Specification and Development in Z and B*. Springer LNCS **3455**, 2005.
15. Graeme Smith and Kirsten Winter. Proving temporal properties of Z specifications using abstraction. *International Conference of Z and B Users (ZB2003)*. Springer LNCS **2651**, 2003.
16. Software Design Group, MIT Computer Science and Artificial Intelligence Laboratory. The Alloy Analyzer, 2007. `http://alloy.mit.edu/`
17. ThyssenKrupp Elevator. TWIN Report, 2005 `http://www.twin.thyssenkrupp-elevator.de/?&L=1`
18. Kirsten Winter and Roger Duke. Model Checking Object-Z using ASM. *Integrated Formal Methods: Third International Conference, IFM 2002*. Springer LNCS **2335**, 2002.
19. The World Wide Web Virtual Library: The Z notation. Tool support, 2005 `http://vl.zuser.org/#tools`